# Octopus Protocol: One-Shot Hardware Discovery and Control for AI Agents via Infrastructure-as-Prompts

Quilee Simeon
MIT
qsimeon@mit.edu

Justin M. Wei
Harvard
mwei@g.harvard.edu

Yile Fan
Harvard
yile_fan@mde.harvard.edu

arXiv:2605.09055v1 [cs.RO] 9 May 2026

***Abstract*—Recent agentic-robotics systems, from Code-as-Policies [1] to modern vision–language–action (VLA) foundation models [2], presuppose that drivers, SDKs, or ROS-style primitives for the target hardware already exist. Writing those primitives is the dominant engineering cost of bringing up new hardware for agent control. We present Octopus Protocol, a system that collapses that cost to a single shell command. Given only raw OS access and a language-model API key, a coding agent executes a five-stage pipeline—PROBE, IDENTIFY, INTERFACE, SERVE, DEPLOY—to discover connected devices, infer their capabilities, generate a Model Context Protocol (MCP) server [3] with typed tools, and deploy it as a live HTTP endpoint. A persistent daemon then monitors the system, heals broken code, and perceives physical state through the camera tools it generated for itself. Two architectural principles make this work: protocols are prompts, not code, and the coding agent is the runtime. We validate the system on three heterogeneous platforms (PC/WSL, Apple Silicon macOS, Raspberry Pi 4) and on a commercial 6-DOF robotic arm with USB camera feedback. One command onboards the hardware in $\sim$10–15 minutes and exposes up to 30 MCP tools; an MCP-compliant client then performs closed-loop visual-motor control through tools no human wrote.**

***Index Terms*—agentic AI, large language models, Model Context Protocol, code generation, embodied AI, robotics infrastructure, self-healing systems, hardware–software co-design**

## I. Scope and Prior Art

Three curves have moved in the same direction for years: software is being written by increasingly autonomous agents; hardware is converging on a small set of standard interfaces (USB, I$^2$C, GPIO, BLE, HTTP); and almost every physical object is becoming electronically addressable. The bottleneck between an AI agent and a piece of hardware is no longer a fundamental gap—it is a glue-code tax: every new device requires a human to read a datasheet, write a driver, integrate it into an SDK, and maintain it across OS updates. Our north star is to eliminate that tax; hardware discovery, interfacing, and control should be as close to one-shot as possible.

Octopus sits at the intersection of three research directions. In agentic robotics, Code-as-Policies [1] showed that LLMs could write robot programs; LeRobot [4] standardized the training stack; and GR00T N1 [2] and related VLA foundation models generalize policies across embodiments. Each composes actions over pre-existing primitives (ROS nodes, Gym APIs, VLA checkpoints). Octopus is the complementary piece: it generates those primitives from first principles. In

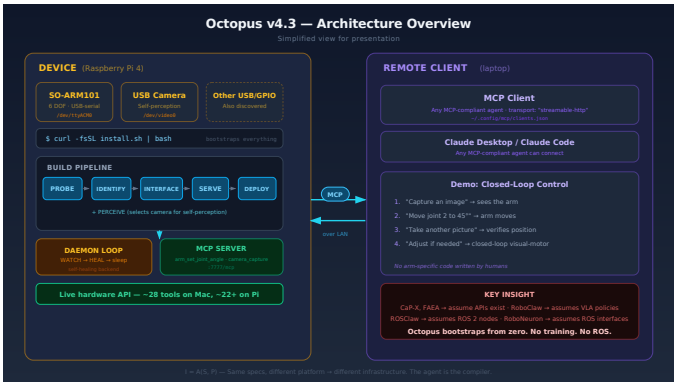


Fig. 1: Octopus architecture. Device (left): a bootstrap command runs a five-stage build pipeline (PROBE, IDENTIFY, INTERFACE, SERVE, DEPLOY) against whatever hardware is plugged in, yielding a live MCP server and a self-healing daemon. Client (right): any MCP-compliant agent drives the hardware through the generated tools.

infrastructure-as-code and its LLM-driven successors, the idea that operational state should be generated from declarative specifications [5] is here extended to a setting where the "infrastructure" is a hardware driver and a live MCP server, not a cloud VM. And in autonomic / self-healing systems [6], [7], the classical vision of systems that monitor and repair themselves finds a concrete embodiment in the Octopus living backend—the same agent that built the driver watches its logs and rewrites broken code.

The architectural claim: if a coding agent can compile a specification to platform-specific hardware code at deploy time, drivers join the list of artifacts that no longer ship as binaries. This reframes the infrastructure layer as a function $I = A(S, P)$, where $A$ is a coding agent, $S$ is a prompt-level specification, and $P$ is the target platform.

## II. Methodology

### A. Five-Stage Build Pipeline

PROBE runs OS-appropriate enumeration (`lsusb`, `system_profiler`, `gpiodetect`) and emits a structured hardware inventory. IDENTIFY maps each vendor/product ID to concrete capabilities (`set_servo_angle`, `capture_image`, ...) via local lookup and web search,

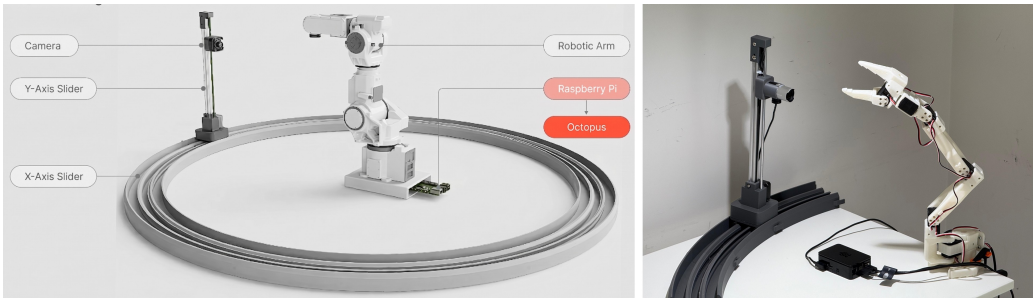


Fig. 2: Hardware. **(a, left)** Design intent: a circular camera-slider rig for multi-view perception (CAD render, aspirational, not yet built). **(b, right)** Current benchtop prototype: off-the-shelf SO-ARM101 arm (Seeed Studio; 6-DOF Feetech STS3215 servos) and a USB camera on a single-axis vertical post atop a partial circular track, driven by a Raspberry Pi 4. Tool count is capped at 30 per installation and varies by platform and what the agent identifies as controllable.

with confidence scoring. INTERFACE emits one MCP tool schema per capability with typed inputs. SERVE writes a complete FastMCP server in which import guards, error handling, and real hardware I/O are generated, not templated. DEPLOY installs dependencies and starts an HTTP/SSE endpoint. The human-written orchestrator is $\sim$640 lines of Python plus $\sim$560 lines of markdown specifications; the generated server is entirely produced at runtime.

### B. Living Backend and Principles

A persistent daemon adds three stages—WATCH, HEAL, PERCEIVE. WATCH tails logs with a cheap model; HEAL prompts the coding agent with failure context to rewrite broken code or reinstall dependencies; PERCEIVE uses the camera tool the system generated for itself, producing a Markov-bounded visual summary (two keyframes plus a natural-language state note) so the agent reasons about the physical world without unbounded context. Two principles separate Octopus from templated integration frameworks: *protocols are prompts, not code*—adding a new hardware class means editing a markdown spec—and *the coding agent is the runtime*—the agent's execution against a spec at deploy time is the software, not a tool for building it.

## III. EVALUATION

*a) Platform portability:* Running the identical markdown specification under the same bootstrap command, we onboarded three heterogeneous hosts: a Windows/WSL PC laptop, an Apple Silicon MacBook Pro, and a Raspberry Pi 4. On each, the coding agent produced a running MCP server with no human intervention. Tool count varied with what was plugged in and what the agent could confidently identify—typically $\sim$18 tools on the Pi and close to the 30-tool cap on the Mac—but the specs and the command were identical. The agent compiled the same intent into three different implementations.

*b) End-to-end task:* On the Pi benchtop rig (Fig. 2b), we demonstrated closed-loop visual-motor control: an MCP client issued natural-language commands to capture an image, observe the arm's pose, move a joint, and verify by a second capture. The perception and control tools were both generated by Octopus in the same run; the client did no hardware-specific reasoning and held no hardware-specific state.

*c) Self-healing:* The HEAL stage fires correctly on induced failures: a missing Python dependency (reinstalled from the log), a hot-unplug/replug of the USB arm (re-probed and regenerated tools), and a deliberate corruption of the generated server (rewritten from the specs). We also pass 14 of 14 integration tests on the orchestrator.

## IV. IMPLICATIONS, OUTLOOK, CONCLUSION

Three consequences follow. *Democratization*: agents on new hardware no longer require driver engineering. *Embodiment*: the agent that produced the hardware interface can perceive through it, closing the sense–act loop without bespoke integration. *Self-sufficiency*: the builder is also the maintainer. Future work extends the protocol to (i) multi-device coordination across a network of Octopus nodes, (ii) networked discovery over Wi-Fi and Bluetooth, (iii) safety-constrained actuation where a generated tool must be proven against a specification before the daemon exposes it, and (iv) the physical camera-slider rig in Fig. 2a. Octopus Protocol shows that an LLM-driven coding agent can bring a previously-unseen device from zero to live, MCP-addressable hardware server in minutes, on commodity computers, with no pre-existing SDK—moving the portable artifact up the stack, from compiled binary to prompt-level specification.

## REFERENCES

[1] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as policies: Language model programs for embodied control," *arXiv preprint arXiv:2209.07753*, 2022.

[2] NVIDIA GEAR Team, "GR00T N1: An open foundation model for generalist humanoid robots," *arXiv preprint arXiv:2503.14734*, 2025.

[3] Anthropic, "Model context protocol," https://modelcontextprotocol.io, 2024.

[4] R. Cadene *et al.*, "LeRobot: State-of-the-art machine learning for real-world robotics in PyTorch," https://github.com/huggingface/lerobot, 2024.

[5] I. Stoica *et al.*, "Specifications: The missing link to making the development of LLM systems an engineering discipline," *arXiv preprint arXiv:2412.05299*, 2024.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] P. Rauba, N. Seedat, K. Kacprzyk, and M. van der Schaar, "Self-healing machine learning: A framework for autonomous adaptation in real-world environments," *arXiv preprint arXiv:2411.00186*, 2024.